Thomas Moskal CSC 344
Professor Graci 05/13/22

The Runtime Stack and the Heap

This essay is a fundamental discussion of what the runtime stack and heap in the field of Computer Science are. The Runtime stack and heap are two structures of computer program memory. In many aspects, they work very differently from each other but, both are necessary for effective computer science programs. Some programmers can create excellent programs with a higher-level programming language without having a full grasp on these two concepts. However, understanding the advantages and disadvantages of these two memory structures is necessary for a programmer to really excel at what they do.

The runtime stack allocates memory at runtime into one contiguous block. The runtime stack, usually just called a stack, uses the stack data structure. This means that the stack works in a last in, first out (LIFO) order. Local non-static variables and parameters are placed into this stack. Once data is taken from the stack with a call to "pop" it, that data is deallocated, and memory is freed up to be re-allocated later. The runtime stack has much faster data retrieval speeds relative to the heap. The memory in the stack is fixed and if a program exceeds the allocated amount of memory the program will crash, this is a common issue known as Stack Overflow.

The Heap is a much larger block of memory relative to the stack. Global variables are stored in the heap. In lower-level programming languages like C and C++ memory must be manually allocated or deallocated by the programmer in the code. If memory is not properly freed up by the user, programs (usually larger ones) may run into an issue known as a memory leak. Some higher-level languages like Python and Java will do garbage collection during run time but this makes the program take longer to execute. The heap is slower and not as guarded as the stack, but it is necessary for dynamic programming.

Explicit Memory Allocation/Deallocation vs Garbage Collection

There are two primary forms of memory management used to prevent issues in a running program, explicit memory allocation/deallocation and garbage collection. Programming languages use these tools in different ways. For example, C, C++ and Haskell allow explicit memory tools for the programmer. On the other hand, Java, Ruby, and Python, use garbage collection in some form. In this essay I will describe why these languages use these tools and how they work.

C, one of the most important and widely used languages in the world allows a programmer to explicitly allocate/deallocate memory. C does this with the Malloc and Calloc functions. C is used for operating systems and embedded software because of this quick and efficient design. Unfortunately, some people can be wary of coding in C because of this type of memory allocation design. If memory is not properly managed by the developer, this can lead to memory leaks which, if severe enough, can possibly crash a computer program. So, explicit memory allocation/deallocation can lead to well-run code when done properly but can also lead to disaster if done improperly.

Garbage collection takes care of memory allocation/deallocation for a user. A language like Java or Python does this automatically which makes them good languages for a beginner to work with if they do not want to be concerned about potential disasters from mismanaged memory and still make great programs. These languages have systems at the compiler level that watches for variables that go out of scope and then deallocate those variables. This makes the language easier to learn and use but has the disadvantage of higher resource overhead.

Rust: Memory Management

1. In C++, we explicitly allocate memory on the heap with **new** and de-allocate it with **delete**. In Rust, we do allocate memory and de-allocate memory at specific points in our program. Thus it doesn't have garbage collection, as Haskell does. But it doesn't work quite the same way as C++.
2. (In Rust ownership) Heap memory always has one owner, and once that owner goes out of scope, the memory gets deallocated.
3. What's cool is that once our string does go out of scope, Rust handles cleaning up the heap memory for it! We don't need to call **delete** as we would in C++. We define memory cleanup for an object by declaring the **drop** function.
4. For people coming from C++ or Java, there seem to be two possibilities. If copying into s2 (a string variable) is a shallow copy, we would expect the sum length to be 12. If it's a deep copy, the sum should be 9. But this code won't compile at all in Rust! The reason is **ownership**.
5. At first, s1 "owns" the heap memory. So when s1 goes out of scope, it will free the memory. But declaring s2 gives over ownership of that memory to the s2 reference. So s1 is now invalid. **Memory can only have one owner.** This is the main idea to get familiar with.
6. Deep copies are often much more expensive than the programmer intends. So a performance-oriented language like Rust avoids using deep copying by default.
7. Like in C++, we can pass a variable by **reference**. We use the ampersand operator (&) for this. It allows another function to "borrow" ownership, rather than "taking" ownership. When it's done, the original reference will still be valid. In this example, the s1 variable re-takes ownership of the memory after the function call ends.
8. This works like a const reference in C++. If you want a *mutable* reference, you can do this as well. The original variable must be mutable, and then you specify **mut** in the type signature.
9. You can only have a single mutable reference to a variable at a time
10. if you want to do a true deep copy of an object, you should use the **clone** function.

Paper Review: Secure PL Adoption and Rust

The article "Benefits and Drawbacks of Adopting a Secure Programming Language: Rust as a Case Study" is a thorough look on the Rust programming language and the opinions of senior software developers (n = 16) and anonymous survey applicants (n = 178) who are familiar with Rust. It goes over the ownership system, blocks, how Rust is used and viewed and how it is changing for the future. Overall, the article gives a good impression on rust and encourages others to adopt it. Part of this drive is because the authors of the article believe if more people become involved in the Rust community, then Rust will become better.

Rust has a mixture of features from high and low-level programming languages as well as a few original features of its own. Rust allows explicit allocation/deallocation of memory like C or C++ but it also automatically calls the drop function at the closing curly bracket to clear out of scope variables. Rust also has a feature called **ownership**, a system based on three rules:

"1. Each value in Rust has a variable that is its owner.

2. There can only be one owner at a time for each value.

3. A value is dropped when its owner goes out of scope."

These rules create a system that make it much easier for someone to prevent common memory management issues that are inherent in a language like C. These rules create a system that is conservative and restrictive, but the developers of Rust know that and have an "escape hatch" of sorts in the form of *unsafe blocks*. These blocks allow a developer to do many things that are unsafe but with experience can be handled safely.

This flexibility is why all but one of the developers interviewed were able to implement Rust at their workplace. Safety and performance were seen by more than 80% of the senior developers as the biggest advantages of Rust. There is however one major issue, the learning curve to Rust is huge. The learning curve is described in the article as "*a near-vertical learning curve"*. Developers recount their experiences in the article as they "*didn't feel fully comfortable with Rust until about three months in, and really solid programming without constantly looking stuff up until about like six months in"*. Rust also lacks critical libraries and infrastructure. The

article ends with the author's listing some steps that they believe will improve adoption or use of Rust itself. This article is a great summary of why someone would be interested in learning Rust for their workplace or personal projects.